

# XQOWL: An Extension of XQuery for OWL Querying and Reasoning

Jesús M. Almendros-Jiménez\*

Dpto. de Informática  
University of Almería  
04120-Almería, SPAIN  
jalmen@ual.es

One of the main aims of the so-called Web of Data is to be able to handle heterogeneous resources where data can be expressed in either XML or RDF. The design of programming languages able to handle both XML and RDF data is a key target in this context. In this paper we present a framework called XQOWL that makes possible to handle XML and RDF/OWL data with XQuery. XQOWL can be considered as an extension of the XQuery language that connects XQuery with SPARQL and OWL reasoners. XQOWL embeds SPARQL queries (via Jena SPARQL engine) in XQuery and enables to make calls to OWL reasoners (HermiT, Pellet and FaCT++) from XQuery. It permits to combine queries against XML and RDF/OWL resources as well as to reason with RDF/OWL data. Therefore input data can be either XML or RDF/OWL and output data can be formatted in XML (also using RDF/OWL XML serialization).

## 1 Introduction

There are two main formats to publish data on the Web. The first format is *XML*, which is based on a tree-based model and for which the *XPath* and *XQuery* languages for querying, and the *XSLT* language for transformation, have been proposed. The second format is *RDF* which is a graph-based model and for which the *SPARQL* language for querying and transformation has been proposed. Both formats (XML and RDF) can be used for describing data of a certain domain of interest. XML is used for instance in the *Dublin Core*<sup>1</sup>, *MPEG-7*<sup>2</sup>, among others, while RDF is used in *DBPedia*<sup>3</sup> and *LinkedLifeData*<sup>4</sup>, among others. The number of organizations that offers their data from the Web is increasing in the last years. The so-called *Linked open data* initiative<sup>5</sup> aims to interconnect the published Web data.

XML and RDF share the same end but they have different data models and query/transformation languages. Some data can be available in XML format and not in RDF format and vice versa. The *W3C* (*World Wide Web Consortium*)<sup>6</sup> proposes transformations from XML data to RDF data (called *lifting*), and vice versa (called *lowering*). RDF has XML-based representations (called *serializations*) that makes possible to represent in XML the graph based structure of RDF. However, XML-based languages are not usually used to query/transform serializations of RDF. Rather than SPARQL is used to query RDF whose

---

\*This work was supported by the EU (FEDER) and the Spanish MINECO Ministry (*Ministerio de Economía y Competitividad*) under grant TIN2013-44742-C4-4-R, as well as by the Andalusian Regional Government (Spain) under Project P10-TIC-6114.

<sup>1</sup><http://www.dublincore.org/>.

<sup>2</sup><http://mpeg.chiariglione.org/>.

<sup>3</sup><http://www.dbpedia.org/>.

<sup>4</sup><http://linkedlifedata.com/>.

<sup>5</sup><http://linkeddata.org/>

<sup>6</sup><http://www.w3.org/>.

syntax resembles SQL and abstract from the XML representation of RDF. The same happens when data are available in XML format: queries and transformations are usually expressed in XPath/XQuery/XSLT, instead of transforming XML to RDF, and using SPARQL.

One of the main aims of the so-called Web of Data is to be able to handle heterogeneous resources where data can be expressed in either XML or RDF. The design of programming languages able to handle both XML and RDF data is a key target in this context and some recent proposals have been presented with this end. One of most known is XSPARQL [6] which is a hybrid language which combines XQuery and SPARQL allowing to query XML and RDF. XSPARQL extends the XQuery syntax with new expressions able to traverse an RDF graph and construct the graph of the result of a query on RDF. One of the uses of XSPARQL is the definition of lifting and lowering from XML to RDF and vice versa. But also XSPARQL is able to query XML and RDF data without transforming them, and obtaining the result in any of the formats. They have defined a formal semantics for XSPARQL which is an extension of the XQuery semantics. The SPARQL2XQuery interoperability framework [5] aims to overcome the same problem by considering as query language SPARQL for both formats (XML and RDF), where SPARQL queries are transformed into XQuery queries by mapping XML Schemas into RDF metadata. In early approaches, SPARQL queries are embedded in XQuery and XSLT [8] and XPath expressions are embedded in SPARQL queries [7].

OWL is an ontology language working with concepts (i.e., classes) and roles (i.e., object/data properties) as well as with individuals (i.e., instances) which fill concepts and roles. OWL can be considered as an extension of RDF in which a richer vocabulary allows to express new relationships. OWL offers more complex relationships than RDF between entities including means to limit the properties of classes with respect to the number and type, means to infer that items with various properties are members of a particular class, and a well-defined model of property inheritance. OWL reasoning [17] is a topic of research of increasing interest in the literature. Most of OWL reasoners (for instance, *HermiT* [12], *Racer* [15], *FaCT++* [18], *Pellet* [16]) are based on tableaux based decision procedures.

In this context, we can distinguish between (1) *reasoning tasks* and (2) *querying tasks* from a given ontology. The most typical (1) *reasoning tasks*, with regard to a given ontology, include: (a) *instance checking*, that is, whether a particular individual is a member of a given concept, (b) *relation checking*, that is, whether two individuals hold a given role, (c) *subsumption*, that is, whether a concept is a subset of another concept, (d) *concept consistency*, that is, consistency of the concept relationships, and (e) a more general case of consistency checking is *ontology consistency* in which the problem is to decide whether a given ontology has a model. However, one can be also interested in (2) *querying tasks* such as: (a) *instance retrieval*, which means to retrieve all the individuals of a given concept, and (b) *property fillers retrieval* which means to retrieve all the individuals which are related to a given individual with respect to a given role.

SPARQL provides mechanisms for querying tasks while OWL reasoners are suitable for reasoning tasks. SPARQL is a query language for RDF/OWL triples whose syntax resembles SQL. OWL reasoners implement a complex deduction procedure including ontology consistency checking that SPARQL is not able to carry out. Therefore SPARQL/OWL reasoners are complementary in the world of OWL.

In this paper we present a framework called XQOWL that makes possible to handle XML and RDF/OWL data with XQuery. XQOWL can be considered as an extension of the XQuery language that connects XQuery with SPARQL and OWL reasoners. XQOWL embeds SPARQL queries (via Jena SPARQL engine) in XQuery and enables to make calls to OWL reasoners (*HermiT*, *Pellet* and *FaCT++*) from XQuery. It permits to combine queries against XML and RDF/OWL resources as well as to reason with RDF/OWL data. Therefore input data can be either XML or RDF/OWL and output data can be formatted in XML (also using RDF/OWL XML serialization). We present two case studies: the first one

which consists on lowering and lifting similar to the presented in [6]; and the second one in which XML analysis is carried out by mapping XML to an ontology and using a reasoner.

Thus the framework proposes to embed SPARQL code in XQuery as well as to make calls to OWL reasoners from XQuery. With this aim a *Java API* has been implemented on top of the OWL API [11] and OWL Reasoner API [10] that makes possible to interconnect XQuery with SPARQL and OWL reasoners. The Java API is invoked from XQuery thanks to the use of the *Java Binding* facility available in most of XQuery processors (this is the case, for instance, of *BaseX* [9], *Exist* [14] and *Saxon* [13]). The Java API enables to connect XQuery to HermiT, Pellet and FaCT++ reasoners as well as to Jena SPARQL engine. The Java API returns the results of querying and reasoning in XML format which can be handled from XQuery. It means that querying and reasoning RDF/OWL with XQOWL one can give XML format to results in either XML or RDF/OWL. In particular, lifting and lowering is possible in XQOWL.

Therefore our proposal can be seen as an extension of the proposed approaches for combining SPARQL and XQuery. Our XQOWL framework is mainly focused on the use of XQuery for querying and reasoning with OWL ontologies. It makes possible to write complex queries that combines SPARQL queries with reasoning tasks. As far as we know our proposal is the first to provide such a combination.

The implementation has been tested with the BaseX processor [9] and can be downloaded from our Web site <http://indalog.ual.es/XQOWL>. There the XQOWL API and the examples of the paper are available as well as installation instructions.

Let us remark that here we continue our previous works on combination of XQuery and the Semantic Web. In [1] we have described how to extend the syntax of XQuery in order to query RDF triples. After, in [2] we have presented a (Semantic Web) library for XQuery which makes possible to retrieve the elements of an ontology as well as to use SWRL. Here, we have followed a new direction, by embedding existent query languages (SPARQL) and reasoners in XQuery.

The structure of the paper is as follows. Section 2 will show an example of OWL ontology used in the rest of the paper as running example. Section 3 will describe XQOWL: the Java API as well as examples of use. Section 4 will present the case study of XML analysis by using an ontology. Finally, Section 5 will conclude and present future work.

## 2 OWL

In this section we show an example of ontology which will be used in the rest of the paper as running example. Let us suppose an ontology about a social network (see Table 1) in which we define ontology classes: *user*, *user\_item*, *activity*; and *event*, *message*  $\sqsubseteq$  *activity* (1); and *wall*, *album*  $\sqsubseteq$  *user\_item* (2). In addition, we can define (object) properties as follows: *created\_by* which is a property whose domain is the class *activity* and the range is *user* (3), and has two sub-properties: *added\_by*, *sent\_by* (4) (used for events and messages, respectively).

We have also *belongs\_to* which is a functional property (5) whose domain is *user\_item* and range is *user* (6); *friend\_of* which is a irreflexive (7) and symmetric (8) property whose domain and range is *user* (9); *invited\_to* which is a property whose domain is *user* and range is *event* (10); *recommended\_friend\_of* which is a property whose domain and range is *user* (11), and is the composition of *friend\_of* and *friend\_of* (12); *replies\_to* which is an irreflexive property (13) whose domain and range is *message* (14); *written\_in* which is a functional property (15) whose domain is *message* and range is *wall* (16); *attends\_to* which is a property whose domain is *user* and range is *event* (17) and is the inverse of the property *confirmed\_by* (18); *i\_like\_it* which is a property whose domain is *user* and

Ontology	
(1) event, message $\sqsubseteq$ activity	(2) wall, album $\sqsubseteq$ user_item
(3) $\forall$ created_by.activity $\sqsubseteq$ user	(4) added_by, sent_by $\sqsubseteq$ created_by
(5) $\top \sqsubseteq \leq 1$ . belongs_to	(6) $\forall$ belongs_to.user_item $\sqsubseteq$ user
(7) $\exists$ friend_of.Self $\sqsubseteq \perp$	(8) friend_of <sup>-</sup> $\sqsubseteq$ friend_of
(9) $\forall$ friend_of.user $\sqsubseteq$ user	(10) $\forall$ invited_to.user $\sqsubseteq$ event
(11) $\forall$ recommended_friend_of.user $\sqsubseteq$ user	(12) friend_of $\cdot$ friend_of $\sqsubseteq$ recommended_friend_of
(13) $\exists$ replies_to.Self $\sqsubseteq \perp$	(14) $\forall$ replies_to.message $\sqsubseteq$ message
(15) $\top \sqsubseteq \leq 1$ . written_in	(16) $\forall$ written_in.message $\sqsubseteq$ wall
(17) $\forall$ attends_to.user $\sqsubseteq$ event	(18) attends_to <sup>-</sup> $\equiv$ confirmed_by
(19) $\forall$ i_like_it.user $\sqsubseteq$ activity	(20) i_like_it <sup>-</sup> $\equiv$ liked_by
(21) $\forall$ content.message $\sqsubseteq$ String	(22) $\forall$ date.event $\sqsubseteq$ DateTime
(23) $\forall$ name.event $\sqsubseteq$ String	(24) $\forall$ nick.user $\sqsubseteq$ String
(25) $\forall$ password.user $\sqsubseteq$ String	(26) event $\sqcap \exists$ confirmed_by.user $\sqsubseteq$ popular
(27) activity $\sqcap \exists$ liked_by.user $\sqsubseteq$ popular	(28) activity $\sqsubseteq \leq 1$ created_by.user
(29) message $\sqcap$ event $\equiv \perp$	

Table 1: Social Network Ontology (in Description Logic Syntax)

range is activity (19), which is the inverse of the property liked\_by (20).

Besides, there are some (data) properties: the content of a message (21), the date (22) and name (23) of an event, and the nick (24) and password (25) of a user. Finally, we have defined the concepts popular which are events confirmed\_by some user and activities liked\_by some user ((26) and (27)) and we have defined constraints: activities are created\_by at most one user (28) and message and event are disjoint classes (29). Let us now suppose the set of individuals and object/data property instances of Table 2.

From OWL reasoning we can deduce new information. For instance, the individual *message1* is an activity, because message is a subclass of activity, and the individual *event1* is also an activity because event is a subclass of activity. The individual *wall\_jesus* is an user\_item because wall is a subclass of user\_item. These inferences are obtained from the subclass relation. In addition, object properties give us more information. For instance, the individuals *message1*, *message2* and *event1* have been created\_by *jesus*, *luis* and *luis*, respectively, since the properties sent\_by and added\_by are sub-properties of created\_by. In addition, the individual *luis* is a friend\_of *jesus* because friend\_of is symmetric. More interesting is that the individual *vicente* is a recommended\_friend\_of *jesus*, because *jesus* is a friend\_of *luis*, and *luis* is a friend\_of *vicente*, which is deduced from the definition of recommended\_friend\_of, which is the composition of friend\_of and friend\_of. Besides, the individual *event1* is confirmed\_by *vicente*, because *vicente* attends\_to *event1* and the properties confirmed\_by and attends\_to are inverses. Finally, there are popular concepts: *event1* and *message2*; the first one has been confirmed\_by *vicente* and the second one is liked\_by *vicente*.

The previous ontology is consistent. The ontology might introduce elements that make the ontology inconsistent. We might add a user being friend\_of of him(er) self. Even more, we can define that certain events and messages are created\_by (either added\_by or sent\_by) more than one user. Also a message can reply to itself. However, there are elements that do not affect ontology consistency. For instance, *event2* has not been created\_by users. The ontology only requires to have at most one creator. Also, messages have not been written\_in a wall.

<i>Ontology Instance</i>
user(jesus), nick(jesus,jalmen), password(jesus,passjesus), friend_of(jesus,luis)
user(luis), nick(luis,lamluis), password(luis,luis0000)
user(vicente), nick(vicente,vicente), password(vicente,vicvicvic), friend_of(vicente,luis), i_like_it(vicente,message2), invited_to(vicente,event1), attends_to(vicente,event1)
event(event1), added_by(event1,luis), name(event1,“Next conference”), date(event1,21/10/2014)
event(event2)
message(message1), sent_by(message1,jesus), content(message1,“I have sent the paper”)
message(message2), sent_by(message2,luis), content(message2,“good luck!”), replies_to(message2,message1)
wall(wall_jesus), belongs_to(wall_jesus,jesus)
wall(wall_luis), belongs_to(wall_luis,luis)
wall(wall_vicente), belongs_to(wall_vicente,vicente)

Table 2: Individuals and object/data properties of the ontology

<i>Java API</i>
public OWLReasoner getOWLReasonerHermiT(OWLOntology ontology)
public OWLReasoner getOWLReasonerPellet(OWLOntology ontology)
public OWLReasoner getOWLReasonerFact(OWLOntology ontology)
public String OWLSPARQL(String filei,String queryStr)
public <T extends OWLAxiom> String OWLQuerySetAxiom(Set<T> axioms)
public <T extends OWLEntity> String[] OWLQuerySetEntity(Set<T> elems)
public <T extends OWLEntity> String[] OWLReasonerNodeEntity(Node <T> elem)
public <T extends OWLEntity> String[] OWLReasonerNodeSetEntity(NodeSet<T> elems)

Table 3: Java API of XQOWL

### 3 XQOWL

XQOWL allows to embed SPARQL queries in XQuery. It also makes possible to make calls to OWL reasoners. With this aim a Java API has been developed.

#### 3.1 The Java API

Now, we show the main elements of the Java API developed for connecting XQuery and SPARQL and OWL reasoners. Basically, the Java API has been developed on top of the OWL API and the OWL Reasoner API and makes possible to retrieve results from SPARQL and OWL reasoners. The elements of the library are shown in Table 3.

The first three elements of the library: *getOWLReasonerHermiT*, *getOWLReasonerPellet* and *getOWLReasonerFact* make possible to instantiate HermiT, Pellet and FaCT++ reasoners. For instance, the code

of *getOWLReasonerHermiT* is as follows:

```
public OWLReasoner getOWLReasonerHermiT(OWLOntology ontology){
    org.semanticweb.HermiT.Reasoner reasoner = new Reasoner(ontology);
    reasoner.precomputeInferences(InferenceType.CLASS_HIERARCHY,
        InferenceType.CLASS_ASSERTIONS,
        ...);
    return reasoner;
};
```

The fourth element of the library *OWLSPARQL* makes possible to instantiate SPARQL Jena engine. The input of this method is an ontology included in a file and a string representing the SPARQL query. The output is a file (name) including the result of the query. The code of *OWLSPARQL* is as follows:

```
public String OWLSPARQL(String filei,String queryStr)
throws FileNotFoundException{
    OntModel model = ModelFactory.createOntologyModel();
    model.read(filei);
    com.hp.hpl.jena.query.Query query = QueryFactory.create(queryStr);
    ResultSet result =
        (ResultSet) SparqlDLExecutionFactory.create(query,model).execSelect();
    String fileName = "./tmp/"+result.hashCode()+"result.owl";
    File f = new File(fileName);
    FileOutputStream file = new FileOutputStream(f);
    ResultSetFormatter.outputAsXML(file,(com.hp.hpl.jena.query.ResultSet) result);
    try { file.close(); } catch (IOException e) {e.printStackTrace();}
    return fileName;
};
```

We can see in the code that the result of the query is obtained in XML format and stored in a file. The rest of elements (i.e, *OWLQuerySetAxiom*, *OWLQuerySetEntity*, *OWLReasonerNodeSetEntity* and *OWLReasonerNodeEntity*) of the Java API make possible to handle the results of calls to SPARQL and OWL reasoners. OWL Reasoners implement Java interfaces of the OWL API for storing OWL elements. The main Java interfaces are *OWLAxiom* and *OWLEntity*. *OWLAxiom* is a Java interface which is a super-interface of all the types of OWL axioms: *OWLSubClassOfAxiom*, *OWLSubDataPropertyOfAxiom*, *OWLSubObjectPropertyOfAxiom*, etc. *OWLEntity* is a Java interface which is a super-interface of all types of OWL elements: *OWLClass*, *OWLDataProperty*, *OWLDatatype*, etc.

The XQOWL API includes the method *OWLQuerySetAxiom* that returns a file name where a set of axioms are included. It also includes *OWLQuerySetEntity* that returns in an array the URI's of a set of entities. Moreover, *OWLReasonerNodeEntity* returns in an array the URI's of a node. Finally, *OWLReasonerNodeSetEntity* returns in an array the URIs of a set of nodes. For instance, the code of *OWLQuerySetEntity* is as follows:

```
public <T extends OWLEntity> String[] OWLQuerySetEntity(Set<T> elems)
{
    String[] result = new String[elems.size()];
    Iterator<T> it = elems.iterator();
    for(int i=0;i<elems.size();i++){
        result[i]=it.next().toStringID();
    };
    return result;
};
```

### 3.2 XQOWL: SPARQL

XQOWL is an extension of the XQuery language. Firstly, XQOWL allows to write XQuery queries in which calls to SPARQL queries are achieved and the results of SPARQL queries in XML format (see

[4]) can be handled by XQuery. In XQOWL, XQuery variables can be bounded to results of SPARQL queries and vice versa, XQuery bounded variables can be used in SPARQL expressions. Therefore, in XQOWL both XQuery and SPARQL queries can share variables.

**Example 3.1** *For instance, the following query returns the individuals of concepts user and event in the social network:*

```
declare namespace spql="http://www.w3.org/2005/sparql-results#";
declare namespace xqo = "java:xqowl.XQOWL";

let $model := "socialnetwork.owl"
for $class in ("sn:user","sn:event")
return
let $queryStr := concat(
  "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  PREFIX sn: <http://www.semanticweb.org/socialnetwork.owl#>
  SELECT ?Ind
  WHERE { ?Ind rdf:type ", $class, " }")
return
let $xqo := xqo:new()
let $res:= xqo:OWLSPARQL($xqo,$model,$queryStr)
return
doc($res)/spql:sparql/spql:results/spql:result/spql:binding/spql:uri/text()
```

Let us observe that the name of the classes (i.e., sn:user and sn:event) is defined by an XQuery variable (i.e., \$class) in a for expression, which is passed as parameter of the SPARQL expression. In addition, the result is obtained in an XQuery variable (i.e. \$res). Here OWLSPARQL of the XQOWL API is used to call the SPARQL Jena engine, which returns a file name (a temporal file) in which the result is found. Now, \$res can be used from XQuery to obtain the URIs of the elements:

```
doc($res)/spql:sparql/spql:results/spql:result/spql:binding/spql:uri/text()
```

*In this case, we obtain the following plain text:*

```
http://www.semanticweb.org/socialnetwork.owl#vicente
http://www.semanticweb.org/socialnetwork.owl#jesus
http://www.semanticweb.org/socialnetwork.owl#luis
http://www.semanticweb.org/socialnetwork.owl#event2
http://www.semanticweb.org/socialnetwork.owl#event1
```

**Example 3.2** *Another example of using XQOWL and SPARQL is the code of lowering from the document:*

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://relations.org">
  <foaf:Person xmlns:foaf="http://xmlns.com/foaf/0.1/" rdf:about="#b1">
    <foaf:name>Alice</foaf:name>
    <foaf:knows>
      <foaf:Person rdf:about="#b4"/>
    </foaf:knows>
    <foaf:knows>
      <foaf:Person rdf:about="#b6"/>
    </foaf:knows>
  </foaf:Person>
  <foaf:Person xmlns:foaf="http://xmlns.com/foaf/0.1/" rdf:about="#b4">
    <foaf:name>Bob</foaf:name>
    <foaf:knows>
      <foaf:Person rdf:about="#b6"/>
    </foaf:knows>
  </foaf:Person>
```

```
<foaf:Person xmlns:foaf="http://xmlns.com/foaf/0.1/" rdf:about="#b6">
  <foaf:name>Charles</foaf:name>
</foaf:Person>
</rdf:RDF>
```

to the document:

```
<relations>
<person name="Alice">
<knows> Bob </knows>
<knows> Charles </knows>
</person>
<person name="Bob">
<knows> Charles </knows>
</person>
<person name="Charles" />
</relations>
```

This example has been taken from [6]<sup>7</sup> in which they show the lowering example in XSPARQL. In our case the code of the lowering example is as follows:

```
declare namespace spql="http://www.w3.org/2005/sparql-results#";
declare namespace xqo = "java:xqowl.XQOWL";
declare variable $model := "relations.rdf";

let $query1 :=
  "PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  SELECT ?Person ?Name
  WHERE {
    ?Person foaf:name ?Name
  } ORDER BY ?Name"
let $xqo := xqo:new(),
$result := xqo:OWLSPARQL($xqo,$model,$query1)
return
for $Binding in doc($result)/spql:sparql/spql:results/spql:result
let $Name := $Binding/spql:binding[@name="Name"]/spql:literal/text(),
  $Person := $Binding/spql:binding[@name="Person"]/spql:uri/text(),
  $PersonName := functx:fragment-from-uri($Person)
return
<person name="{ $Name }">{
let $query2 :=
  concat(
    "PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
    PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX rel: <http://relations.org#>
    PREFIX foaf: <http://xmlns.com/foaf/0.1/>
    SELECT ?FName
    WHERE {
      _:",$PersonName," foaf:knows ?Friend .
      _:",$PersonName," foaf:name ", "'", $Name, "' .
      ?Friend foaf:name ?FName
    }")
let $result2 := xqo:OWLSPARQL($xqo,$model,$query2)
return
for $FName in doc($result2)/spql:sparql/spql:results/spql:result/spql:binding/
  spql:literal/text()
return
<knows>{ $FName }</knows>
```

<sup>7</sup>XSPARQL works with blank nodes, and there the RDF document includes nodeID tag for each RDF item. In XQOWL we cannot deal with blank nodes at all, and therefore a preprocessing of the RDF document is required: nodeID tags are replaced by about.



```

}
</person>
}
</relations>

```

In this example, two SPARQL queries are nested and share variables. The result of the first SPARQL query (i.e., \$PersonName and \$Name) is used in the second SPARQL query.

### 3.3 XQOWL: OWL Reasoners

XQOWL can be also used for querying and reasoning with OWL. With this aim the OWL API and OWL Reasoner API have been integrated in XQuery. Also for this integration, the XQOWL API is required. For using OWL Reasoners from XQOWL there are some calls to be made from XQuery code. Firstly, we have to instantiate the ontology manager by using *createOWLOntologyManager*; secondly, the ontology has to be loaded by using *loadOntologyFromOntologyDocument*; thirdly, in order to handle OWL elements we have to instantiate the data factory by using *getOWLDataFactory*; finally, in order to select a reasoner *getOWLReasonerHermiT*, *getOWLReasonerPellet* and *getOWLReasonerFact* are used.

**Example 3.3** For instance, we can query the object properties of the ontology using the OWL API as follows:

```

let $xqo := xqo:new(),
    $man := api:createOWLOntologyManager(),
    $fileName := file:new($file),
    $ont := om:loadOntologyFromOntologyDocument($man,$fileName)
return
doc(xqo:OWLQuerySetAxiom($xqo,o:getAxioms($ont)))/rdf:RDF/owl:ObjectProperty

```

obtaining the following result:

```

<ObjectProperty...rdf:about="...#added_by">
  <rdfs:subPropertyOf rdf:resource="...#created_by"/>
  <rdfs:domain rdf:resource="...#event"/>
  <rdfs:range rdf:resource="...#user"/>
</ObjectProperty>
<ObjectProperty ... rdf:about="...#attends_to">
  <inverseOf rdf:resource="...#confirmed_by"/>
  <rdfs:range rdf:resource="...#event"/>
  <rdfs:domain rdf:resource="...#user"/>
</ObjectProperty>
...

```

**Example 3.4** Another example of query using the OWL API is the following which requests class axioms related to wall and event:

```

let $xqo := xqo:new(),
    $man := api:createOWLOntologyManager(),
    $fileName := file:new($file),
    $ont := om:loadOntologyFromOntologyDocument($man,$fileName),
    $fact := om:getOWLDataFactory($man)
return
for $class in ("wall","event")
let $iri := iri:create(concat($base,$class)),
    $class := df:getOWLClass($fact,$iri)
return
doc(xqo:OWLQuerySetAxiom($xqo,o:getAxioms($ont,$class)))/rdf:RDF/owl:Class

```

in which a for expression is used to define the names of the classes to be retrieved, obtaining the following result:

```

<Class ... rdf:about="...#user_item"/>
<Class ... rdf:about="...#wall">
  <rdfs:subClassOf rdf:resource="...#user_item"/>
</Class>
<Class ... rdf:about="...#activity"/>
<Class ... rdf:about="...#event">
  <rdfs:subClassOf rdf:resource="...#activity"/>
  <disjointWith rdf:resource="...#message"/>
</Class>
<Class ... rdf:about="...#message"/>

```

Now we can see examples about how to use XQOWL for reasoning with an ontology. With this aim, we can use the OWL Reasoner API (as well as the XQOWL API). The XQOWL API allows easily to use HermiT, Pellet and FaCT++ reasoners.

**Example 3.5** *For instance, let us suppose we want to check the consistence of the ontology by the HermiT reasoner. The code is as follows:*

```

let $xqo := xqo:new(),
    $man := api:createOWLOntologyManager(),
    $fileName := file:new($file),
    $ont := om:loadOntologyFromOntologyDocument($man,$fileName),
    $fact := om:getOWLDataFactory($man),
    $reasoner := xqo:getOWLReasonerHermiT($xqo,$ont),
    $boolean := r:isConsistent($reasoner),
    $dispose := r:dispose($reasoner)
return $boolean

```

*which returns true. Here the HermiT reasoner is instantiated by using getOWLReasonerHermiT. In addition, the OWL Reasoner API method isConsistent is used to check ontology consistence. Each time the work of the reasoner is done, a call to dispose is required.*

**Example 3.6** *Let us suppose now we want to retrieve instances of concepts activity and user. Now, we can write the following query using the HermiT reasoner:*

```

for $classes in ("activity","user")
let $xqo := xqo:new(),
    $man := api:createOWLOntologyManager(),
    $fileName := file:new($file),
    $ont := om:loadOntologyFromOntologyDocument($man,$fileName),
    $fact := om:getOWLDataFactory($man),
    $iri := iri:create(concat($base,$classes)),
    $reasoner := xqo:getOWLReasonerHermiT($xqo,$ont),
    $class := df:getOWLClass($fact,$iri),
    $result:= r:getInstances($reasoner,$class,false()),
    $dispose := r:dispose($reasoner)
return
<concept class="{ $classes }">
{ for $instances in xqo:OWLReasonerNodeSetEntity($xqo,$result)
  return <instance>{substring-after($instances,'#')}</instance>}
</concept>

```

*obtaining the following result in XML format:*

```

<concept class="activity">
  <instance>message1</instance>
  <instance>message2</instance>
  <instance>event1</instance>
  <instance>event2</instance>
</concept>

```

```
<concept class="user">
  <instance>jesus</instance>
  <instance>vicente</instance>
  <instance>luis</instance>
</concept>
```

Here `getInstances` of the OWL Reasoner API is used to retrieve the instances of a given ontology class. In addition, a call to `create` of the OWL API, which creates the IRI of the class, and a call to `getClass` of the OWL API, which retrieves the class, are required. The OWL Reasoner is able to deduce that `message1` and `message2` belong to concept `activity` since they belong to concept `message` and `message` is a subconcept of `activity`. The same can be said for events.

**Example 3.7** Let us suppose now we want to retrieve the subconcepts of `activity` using the Pellet reasoner. The code is as follows:

```
let $xqo := xqo:new(),
    $man := api:createOWLOntologyManager(),
    $fileName := file:new($file),
    $ont := om:loadOntologyFromOntologyDocument($man,$fileName),
    $fact := om:getOWLDDataFactory($man),
    $iri := iri:create(concat($base,"activity")),
    $reasoner := xqo:getOWLReasonerPellet($xqo,$ont),
    $class := df:getOWLClass($fact,$iri),
    $result:= r:getSubClasses($reasoner,$class,false()),
    $dispose := r:dispose($reasoner)
return
for $subclass in xqo:OWLReasonerNodeSetEntity($xqo,$result)
  return <subclass>{substring-after($subclass,'#')} </subclass>
```

and the result in XML format is as follows:

```
<subclass>popular_message</subclass>
<subclass>event</subclass>
<subclass>Nothing</subclass>
<subclass>popular_event</subclass>
<subclass>message</subclass>
```

Here `getSubClasses` of the OWL Reasoner API is used.

**Example 3.8** Finally, let us suppose we want to retrieve the recommended friends of `jesus`. Now, the query is as follows:

```
let $xqo := xqo:new(),
    $man := api:createOWLOntologyManager(),
    $fileName := file:new($file),
    $ont := om:loadOntologyFromOntologyDocument($man,$fileName),
    $fact := om:getOWLDDataFactory($man),
    $iri := iri:create(concat($base,"recommended_friend_of")),
    $iri2 := iri:create(concat($base,"jesus")),
    $reasoner := xqo:getOWLReasonerPellet($xqo,$ont),
    $property := df:getOWLObjectProperty($fact,$iri),
    $ind := df:getOWLNamedIndividual($fact,$iri2),
    $result:= r:getObjectPropertyValues($reasoner,$ind,$property),
    $dispose := r:dispose($reasoner)
return
for $rfriend in xqo:OWLReasonerNodeSetEntity($xqo,$result)
  return
<recommended_friend>
  {substring-after($rfriend,'#')}
</recommended_friend>
```

and the answer as follows:

```
<recommended_friend>jesus</recommended_friend>
<recommended_friend>vicente</recommended_friend>
```

Here the OWL Reasoner API is used to deduce the friends of friends of *jesus*. Due to symmetry of friend relationship, a person is a recommended friend of itself.

## 4 Using XQOWL for XML Analysis

Now, we show an example in which XQOWL is used to analyze the semantic content of an XML document. This example was used in our previous work [3] to illustrate the use of our Semantic Web library for XQuery. The example takes an XML document as input as follows:

```
<?xml version='1.0'?>
<conference>
  <papers>
    <paper id="1" studentPaper="true">
      <title> XML Schemas </title>
      <wordCount> 1200 </wordCount>
    </paper>
    <paper id="2" studentPaper="false">
      <title> XML and OWL </title>
      <wordCount> 2800 </wordCount>
    </paper>
    <paper id="3" studentPaper="true">
      <title> OWL and RDF </title>
      <wordCount> 12000 </wordCount>
    </paper>
  </papers>
  <researchers>
    <researcher id="a" isStudent="false" manuscript="1" referee="1">
      <name>Smith </name>
    </researcher>
    <researcher id="b" isStudent="true" manuscript="1" referee="2">
      <name>Douglas </name>
    </researcher>
    <researcher id="c" isStudent="false" manuscript="2" referee="3">
      <name>King </name>
    </researcher>
    <researcher id="d" isStudent="true" manuscript="2" referee="1">
      <name>Ben </name>
    </researcher>
    <researcher id="e" isStudent="false" manuscript="3" referee="3">
      <name>William </name>
    </researcher>
  </researchers>
</conference>
```

The document lists *papers* and *researchers* involved in a *conference*. Each *paper* and *researcher* has an identifier (represented by the attribute *id*), and has an associated set of labels: *title* and *wordCount* for *papers* and *name* for *researchers*. Furthermore, they have attributes *studentPaper* for *papers* and *isStudent*, *manuscript* and *referee* for *researchers*. The meaning of *manuscript* and *referee* is that the given researcher has submitted the paper of number described by *manuscript* as well as has participated as reviewer of the paper of number given by *referee*.

Now, let us suppose that we would like to analyze the content of the XML document in order to detect constraints which are violated. In particular, the revision system of the conference forbids that an student is a reviewer as well as a research is a reviewer of his(her) own paper.

In order to analyze the document the idea is to create an ontology to represent the same elements of the XML document. This ontology contains in the **TBox** a vocabulary to represent submissions. It includes class names *Paper* and *Researcher*. But also it includes *PaperofSenior*, *PaperofStudent*, *Student* and *Senior*. The individuals of *PaperofSenior* are the papers for which *studentPaper* of the XML document has been set to false. The individuals of *PaperofStudent* are the papers for which *studentPaper* of the XML document has been set to true. Analogously, the individuals of *Senior* and *Student* are the researchers for which *isStudent* has been set to false, respectively, to true. In addition the ontology includes object properties *manuscript* and *referee*, and data properties *wordCount*, *name* and *title*.

Now, the idea is to express the revision system constraints as constraints of the ontology. Thus, the ontology includes two restrictions to be checked: *Student* and *Reviewer* classes are disjoint while *manuscript* and *referee* are disjoint object properties.

In order to analyze a given XML document, we can use XQOWL with two ends.

- To transform the XML document to the ontology **ABox**.
- To check consistence of the ontology.

The code of the transformation to the ontology **ABox** is as follows:

```
let $name := /conference
let $ontology1 :=
  (for $x in $name/papers/paper return
    sw:toClassFiller(sw:ID($x/@id), "#Paper") union
    (
      let $studentPaper:= $x/@studentPaper return
      if (data($studentPaper)="true") then
        sw:toClassFiller(sw:ID($x/@id), "#PaperofStudent")
      else sw:toClassFiller(sw:ID($x/@id), "#PaperofSenior")
    ) union
    sw:toDataFiller(sw:ID($x/@id), "title", $x/title, "string") union
    sw:toDataFiller(sw:ID($x/@id), "wordCount", $x/wordCount, "integer")
  )
let $ontology2 :=
  (for $y in $name/researchers/researcher return
    sw:toClassFiller(sw:ID($y/@id), "#Researcher") union
    sw:toDataFiller(sw:ID($y/@id), "name", $y/name, "string") union
    (
      let $student:= $y/@isStudent return
      if (data($student)="true") then
        sw:toClassFiller(sw:ID($y/@id), "#Student")
      else sw:toClassFiller(sw:ID($y/@id), "#Senior")
    ) union
    sw:toObjectFiller(sw:ID($y/@id), "manuscript", sw:ID($y/@manuscript)) union
    sw:toObjectFiller(sw:ID($y/@id), "referee", sw:ID($y/@referee))
  )
return
let $mapping := $ontology1 union $ontology2
return
let $doc :=
document{
  <rdf:RDF ...>
    {doc("ontology_papers.owl")/rdf:RDF/*}
    {$mapping}
  </rdf:RDF>
}
```

Here we have used the Semantic Web library for XQuery defined in [3]. Basically, we have created the instance of the ontology by using *sw:toClassFiller*, *sw:toDataFiller* and *sw:toObjectFiller* which make possible to create instances of classes, data and object properties, respectively. At the end of the code, the ontology **TBox** is incorporated (which is stored in the file "*ontology\_papers.owl*"). Now, the

consistence checking using the Hermit reasoner is as follows, where \$doc is the result of the previous query:

```
let $xqo := xqo:new(),
$man := api:createOWL0ntologyManager(),
$seq := file:write("ontology_analysis.owl",$doc),
$fileName := file_io:new($file),
$ont := om:loadOntologyFromOntologyDocument($man,$fileName),
$fact := om:getOWLDataFactory($man),
$reasoner := xqo:getOWLReasonerHermit($xqo,$ont),
$boolean := r:isConsistent($reasoner),
$dispose := r:dispose($reasoner)
return $boolean
```

## 5 Conclusions and Future Work

In this paper we have presented an extension of XQuery called XQOWL to query XML and RDF/OWL documents, as well as to reason with RDF/OWL resources. We have described the XQOWL API that allows to make calls from XQuery to SPARQL and OWL Reasoners. Also we have shown examples of use of XQOWL. The main advantage of the approach is to be able to handle both types of documents through the sharing of variables between XQuery and SPARQL/OWL Reasoners. The implementation has been tested with the BaseX processor [9] and can be downloaded from our Web site <http://indalog.ual.es/XQOWL>. As future work, we would like to extend our work as follows. Firstly, we would like to extend our Java API. More concretely, with the SWRL API in order to execute rules from XQuery, and to be able to provide explanations about ontology inconsistency. Secondly, we would like to use our framework in ontology transformations (refactoring, coercion, splitting, amalgamation) and matching.

## References

- [1] Jesús Manuel Almendros-Jiménez (2009): *Extending XQuery for Semantic Web Reasoning*. In Salvador Abreu & Dietmar Seipel, editors: *Applications of Declarative Programming and Knowledge Management - 18th International Conference, INAP 2009, Évora, Portugal, November 3-5, 2009, Revised Selected Papers, Lecture Notes in Computer Science* 6547, Springer, pp. 117–134, doi:10.1007/978-3-642-20589-7\_8.
- [2] Jesús Manuel Almendros-Jiménez (2011): *Querying and Reasoning with RDF(S)/OWL in XQuery*. In Xiaoyong Du, Wenfei Fan, Jianmin Wang, Zhiyong Peng & Mohamed A. Sharaf, editors: *Web Technologies and Applications - 13th Asia-Pacific Web Conference, APWeb 2011, Beijing, China, April 18-20, 2011. Proceedings, Lecture Notes in Computer Science* 6612, Springer, pp. 450–459, doi:10.1007/978-3-642-20291-9\_51.
- [3] Jesús Manuel Almendros-Jiménez (2012): *Using OWL and SWRL for the Semantic Analysis of XML Resources*. In Robert Meersman, Hervé Panetto, Tharam S. Dillon, Stefanie Rinderle-Ma, Peter Dadam, Xiaofang Zhou, Siani Pearson, Alois Ferscha, Sonia Bergamaschi & Isabel F. Cruz, editors: *On the Move to Meaningful Internet Systems: OTM 2012, Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2012, Rome, Italy, September 10-14, 2012. Proceedings, Part II, Lecture Notes in Computer Science* 7566, Springer, pp. 915–931, doi:10.1007/978-3-642-33615-7\_33.
- [4] Dave Beckett & Jeen Broekstra (2013): *SPARQL Query Results XML Format (Second Edition)*. <http://http://www.w3.org/TR/rdf-sparql-XMLres/>.
- [5] Nikos Bikakis, Chrisa Tsinaraki, Ioannis Stavrakantonakis, Nektarios Gioldasis & Stavros Christodoulakis (2014): *The SPARQL2XQuery interoperability framework*. *World Wide Web*, pp. 1–88, doi:10.1007/s11280-013-0257-x.

- [6] Stefan Bischof, Stefan Decker, Thomas Krennwallner, Nuno Lopes & Axel Polleres (2012): *Mapping between RDF and XML with XSPARQL*. *Journal on Data Semantics* 1(3), pp. 147–185, doi:10.1007/s13740-012-0008-7.
- [7] Matthias Droop, Markus Flarer, Jinghua Groppe, Sven Groppe, Volker Linnemann, Jakob Pinggera, Florian Santner, Michael Schier, Felix Schöpf & Hannes Staffler (2009): *Bringing the XML and semantic web worlds closer: transforming XML into RDF and embedding XPath into SPARQL*. In: *Enterprise Information Systems*, Springer, pp. 31–45, doi:10.1007/978-3-642-00670-8\_3.
- [8] Sven Groppe, Jinghua Groppe, Volker Linnemann, Dirk Kukulenz, Nils Hoeller & Christoph Reinke (2008): *Embedding SPARQL into XQUERY/XSLT*. In: *Proceedings of the 2008 ACM symposium on Applied computing*, ACM, pp. 2271–2278, doi:10.1145/1363686.1364228.
- [9] Christian Grun (2014): *BaseX. The XML Database*. <http://basex.org>.
- [10] Matthew Horridge (2009): *OWL Reasoner API*. <http://owlapi.sourceforge.net/javadoc/org/semanticweb/owlapi/reasoner/OWLReasoner.html>.
- [11] Matthew Horridge & Sean Bechhofer (2011): *The OWL API: A Java API for OWL Ontologies*. *Semant. web* 2(1), pp. 11–21. Available at <http://dl.acm.org/citation.cfm?id=2019470.2019471>.
- [12] Ian Horrocks, Boris Motik & Zhe Wang (2012): *The HermiT OWL Reasoner*. In Ian Horrocks, Mikalai Yatskevich & Ernesto Jiménez-Ruiz, editors: *Proceedings of the 1st International Workshop on OWL Reasoner Evaluation (ORE-2012)*, Manchester, UK, July 1st, 2012, *CEUR Workshop Proceedings* 858, CEUR-WS.org. Available at [http://ceur-ws.org/Vol-858/ore2012\\_paper13.pdf](http://ceur-ws.org/Vol-858/ore2012_paper13.pdf).
- [13] Michael Kay (2008): *Ten Reasons Why Saxon XQuery is Fast*. *IEEE Data Eng. Bull.* 31(4), pp. 65–74. Available at <http://sites.computer.org/debull/A08dec/saxonica.pdf>.
- [14] Wolfgang Meier (2003): *eXist: An open source native XML database*. In: *Web, Web-Services, and Database Systems*, Springer, pp. 169–183, doi:10.1007/3-540-36560-5\_13.
- [15] Ralf Möller, Volker Haarslev & Sebastian Wandelt (2008): *The Revival of Structural Subsumption in Tableau-based Reasoners*. In Franz Baader, Carsten Lutz & Boris Motik, editors: *Proceedings of the 21st International Workshop on Description Logics (DL2008)*, Dresden, Germany, May 13–16, 2008, *CEUR Workshop Proceedings* 353, CEUR-WS.org. Available at <http://ceur-ws.org/Vol-353/MoellerHaarslevWandelt.pdf>.
- [16] Evren Sirin, Bijan Parsia, Bernardo C. Grau, Aditya Kalyanpur & Yarden Katz (2007): *Pellet: A practical OWL-DL reasoner*. *Web Semantics: Science, Services and Agents on the World Wide Web* 5(2), pp. 51–53, doi:10.1016/j.websem.2007.03.004.
- [17] Steffen Staab & Rudi Studer (2010): *Handbook on ontologies*. Springer, doi:10.1007/978-3-540-92673-3.
- [18] Dmitry Tsarkov & Ian Horrocks (2006): *FaCT++ Description Logic Reasoner: System Description*. In Ulrich Furbach & Natarajan Shankar, editors: *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17–20, 2006, Proceedings, Lecture Notes in Computer Science* 4130, Springer, pp. 292–297, doi:10.1007/11814771\_26.